COMPSCI 110 Part 2 : A RoadMap

1. Week 7 : The Compilation Process
   Chapter 11 (Textbook)
2. Week 8 : Operating Systems
   Chapter 6 (Textbook) – Section 6.4 onwards
3. Week 9 : Computer Networks
   Chapter 7 (Textbook)
4. Week 10 : Computer Security
   Chapter 8 (Textbook)
5. Week 11 : Turing Machines
   Chapter 12 (Textbook)
6. Week 12 : Artificial Intelligence
   Chapter 15 (Textbook)

COMPILERS AND LANGUAGE TRANSLATION

CHAPTER 11

https://xkcd.com/303

LEARNING OBJECTIVES

The case for high-level programming languages

List the phases of a typical compiler and describe the purpose of each phase

Demonstrate how to break up a string of text into tokens.

Understand grammar rules written in BNF and use them to parse statements.

Explain how semantic analysis uses semantic records to determine meaning

Show what a code generator does.

THE CASE FOR HIGH-LEVEL PROGRAMMING LANGUAGES

You have learnt how to write simple programs in assembly language.

Nowadays, almost nobody writes code in assembly language. Why?

Problems with Assembly language

- We need to bring our data into a small number of registers and manipulate it there, then store it back in memory again.
- Programming even rather simple operations becomes quite repetitive in nature.
- It revolves around the needs of the hardware rather than around the algorithm.

So, we usually write code in a high-level programming language (e.g. Python or Java).

- There is a need to translate the high-level programming code to machine code.
- Such a program is called a **translator** program (or, a programming language processor).

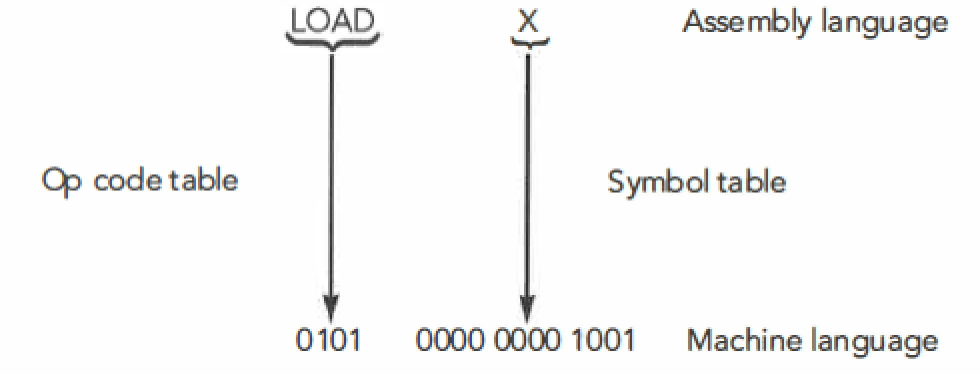
Assemblers Vs Compilers

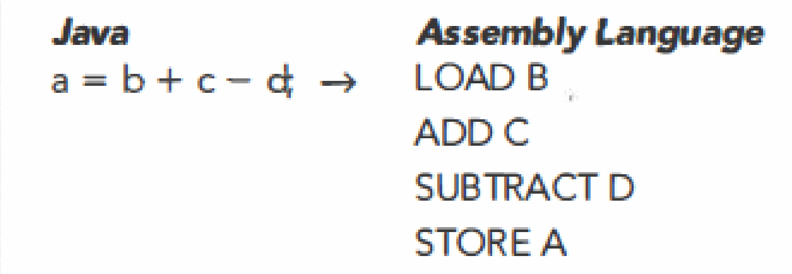
An assembler is a program that converts an assembly language program to machine code

A compiler (or, translator) is a program that converts a high-level language program to machine code

Assembly language and machine language are related one-to-one.

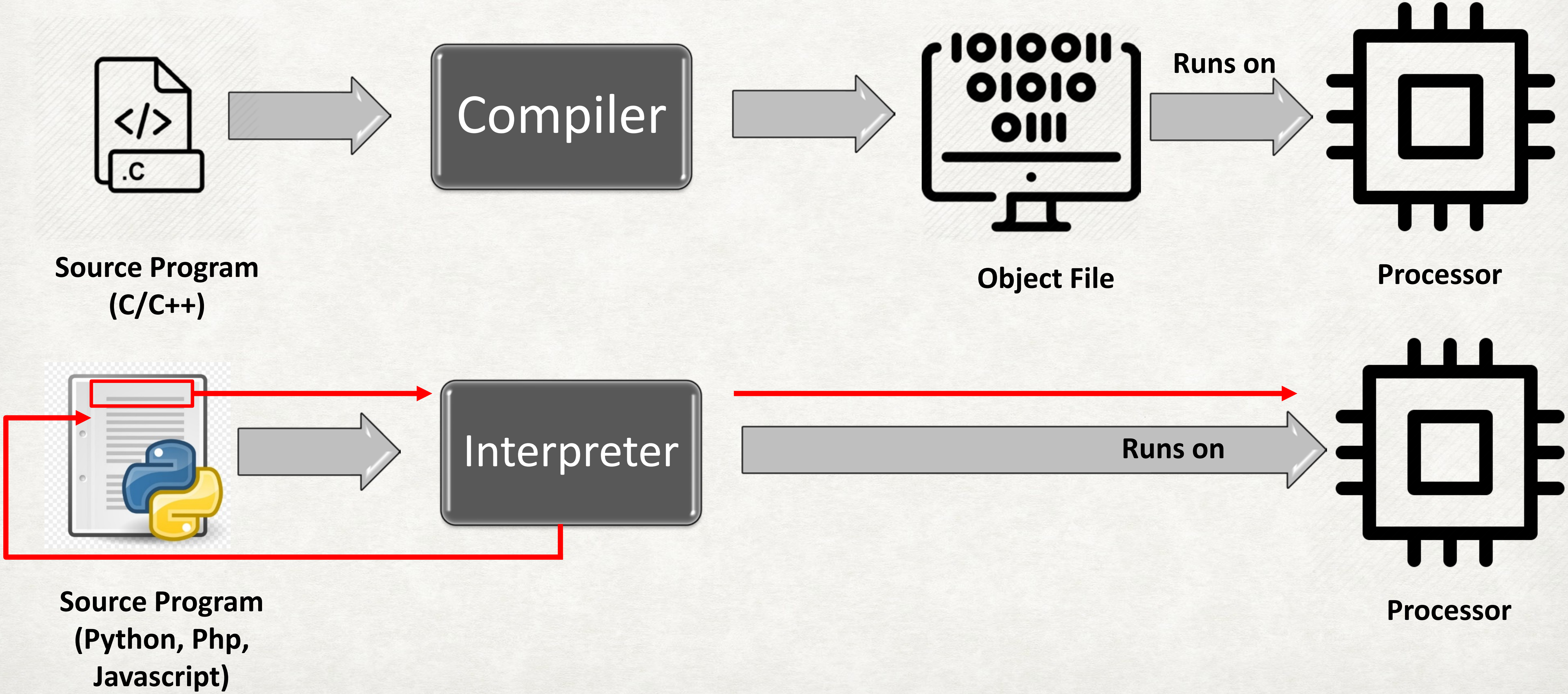
Relationship between high level language and machine language is one to many.





- To determine which machine language instruction must be generated, a translator cannot simply look up a name in the table.
- It must do a thorough linguistic analysis of the structure (**syntax**) and meaning (**semantics**) of each high-level language statement before deciding what to do.

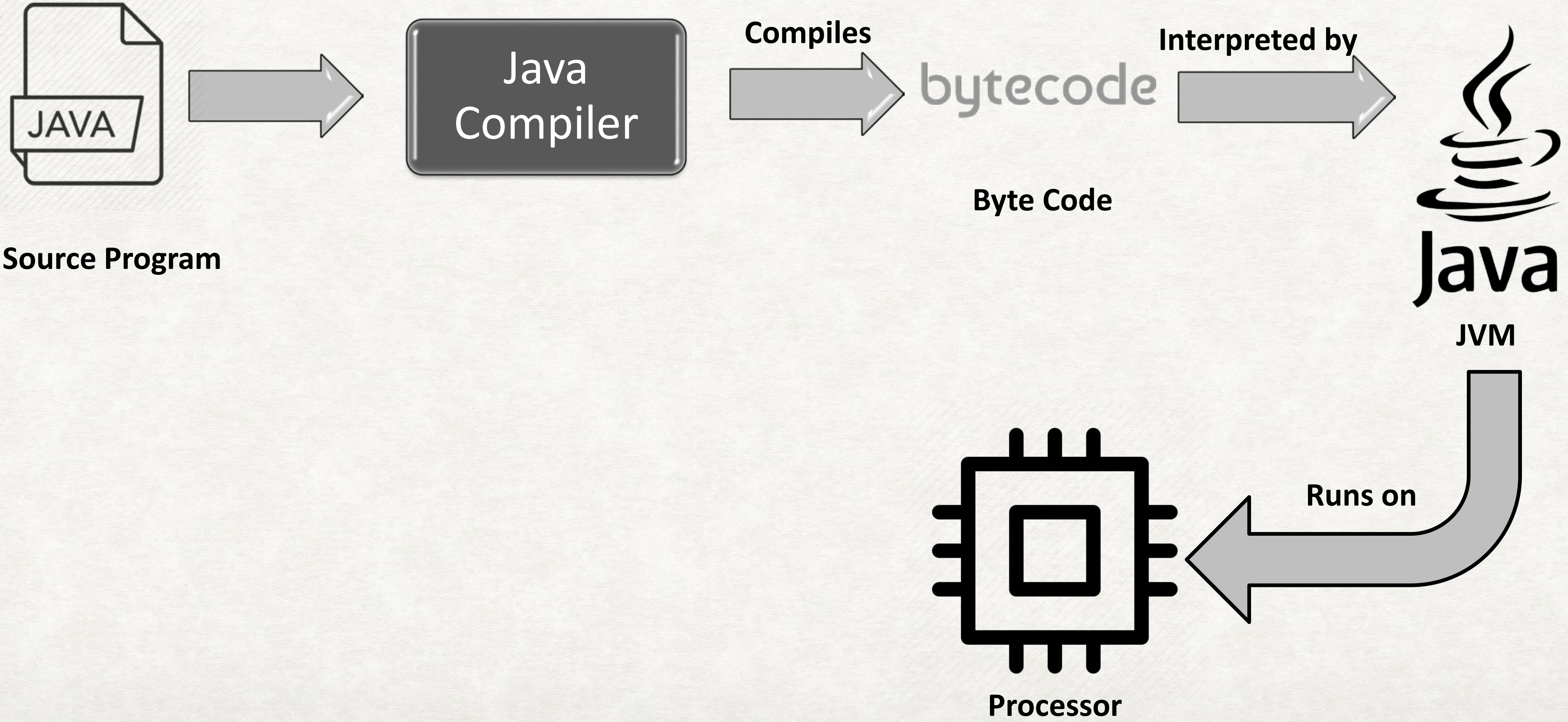
TWO TYPES OF TRANSLATORS : COMPILERS AND INTERPRETERS



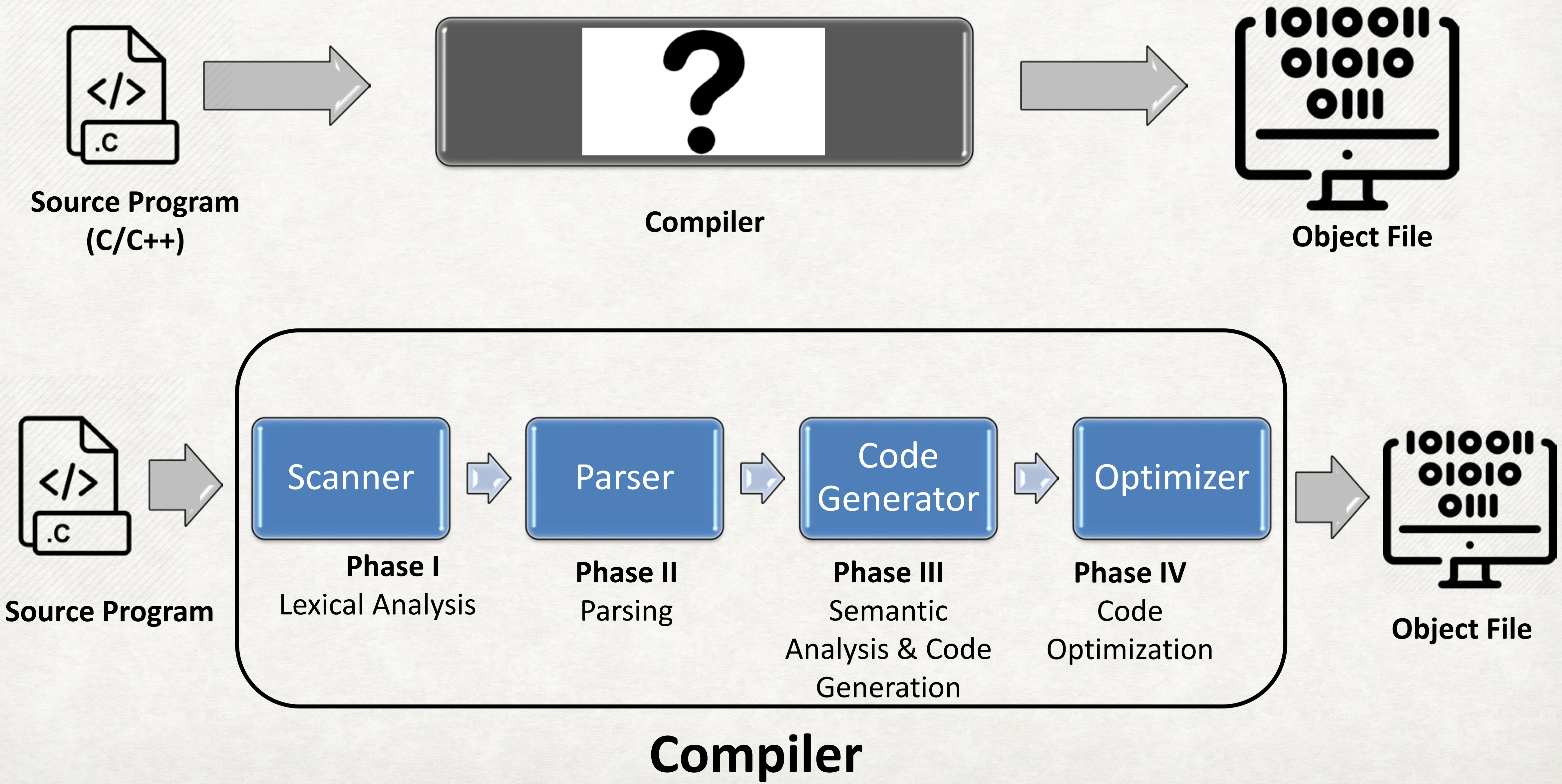
Compiler Vs Interpreter

Compiler	Interpreter
Ouput : Compiled executable file. To get the results, we need to run the file.	Output : result of execution of the code. It therefore runs slower than the compiled code.
Generates intermediate object code and thus requires more memory.	No intermediate code required and thus needs less memory.
Less flexible. A compiler will run only once. If we make changes to the source code, we need to invoke the compiler again.	More Flexible. An interpreter runs again and again and will reinterpret source code when it changes.
Safer to Distribute. The source code is not required after compilation.	An interpreter requires the source code in order to translate and execute the program every single time.
Harder to Debug. Errors surface only after the entire program has been translated.	Easier to Debug. Errors arise as the code is being interpreted.

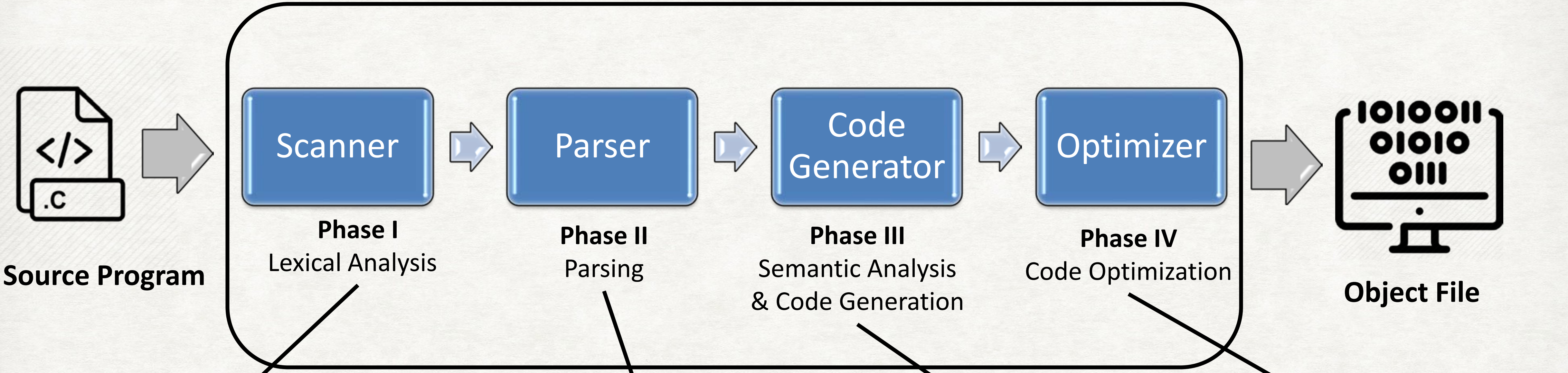
MIXED APPROACHES



HOW DOES A COMPILER WORK?



FOUR PHASES OF COMPILATION



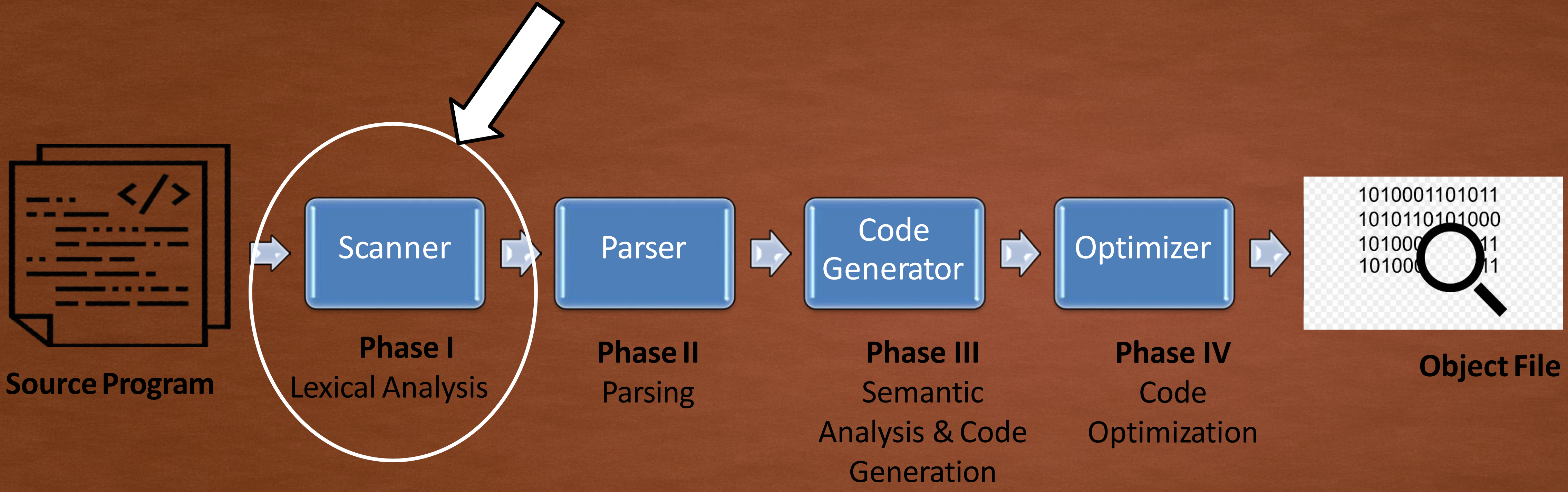
- Groups characters into "**tokens**".
- Equivalent to grouping letters into words in natural language.

- Checks to see whether the tokens are syntactically correct according to the rules (or, "**grammar**") of the programming language.
- Equivalent to checking whether words are connected together to make a grammatically correct sentence

- Analyzes the meaning of each statement and generates machine instructions to execute the statement.
- Does a "**Semantic**" check.

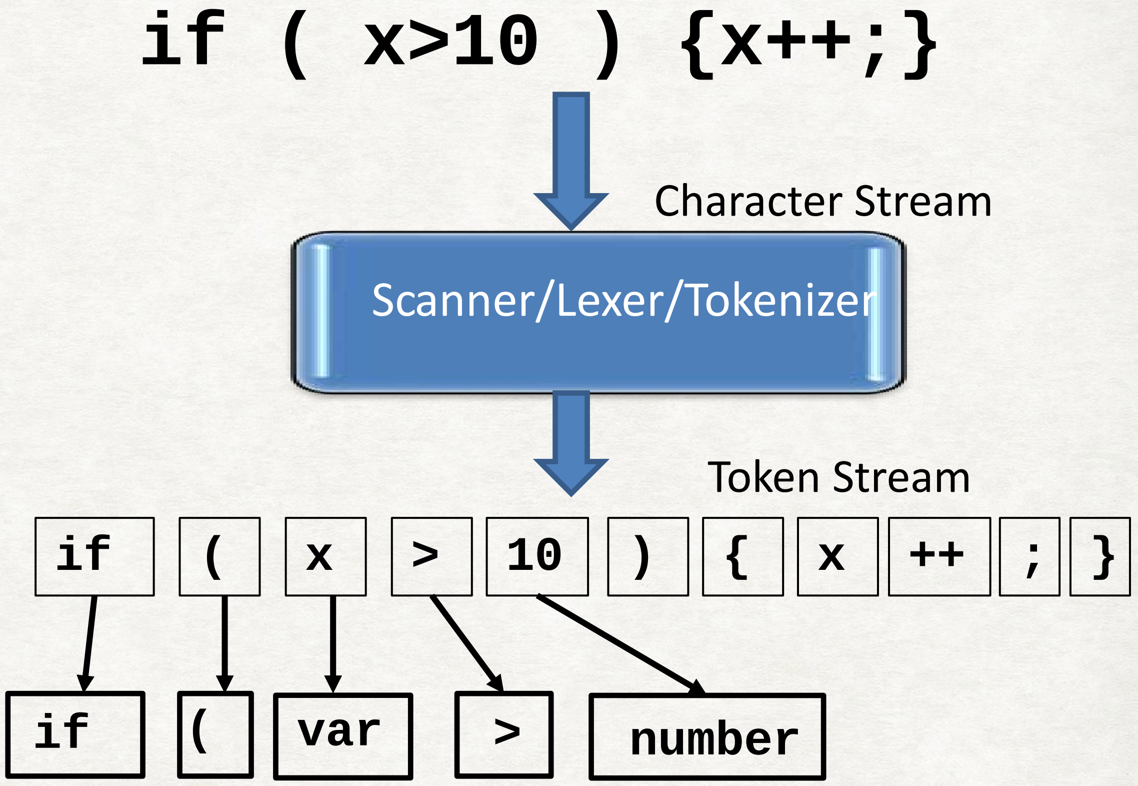
Improves efficiency of code in time or space or both.

Phase I : Lexicographical Analysis



PHASE I : LEXICOGRAPHICAL ANALYSIS

- The compiler takes a piece of source code in a programming language and breaks it up into a series of *tokens*.
  - aka *lexer, tokenizer* or *scanner.*
- Tokens are like the vocabulary of the source language.
  - Equivalent to words in the English Language.
- Discards unnecessary characters
  Examples: blanks, tabs, and comment text
- Determines the type of each **token**
  Examples: symbol, number, and left parenthesis

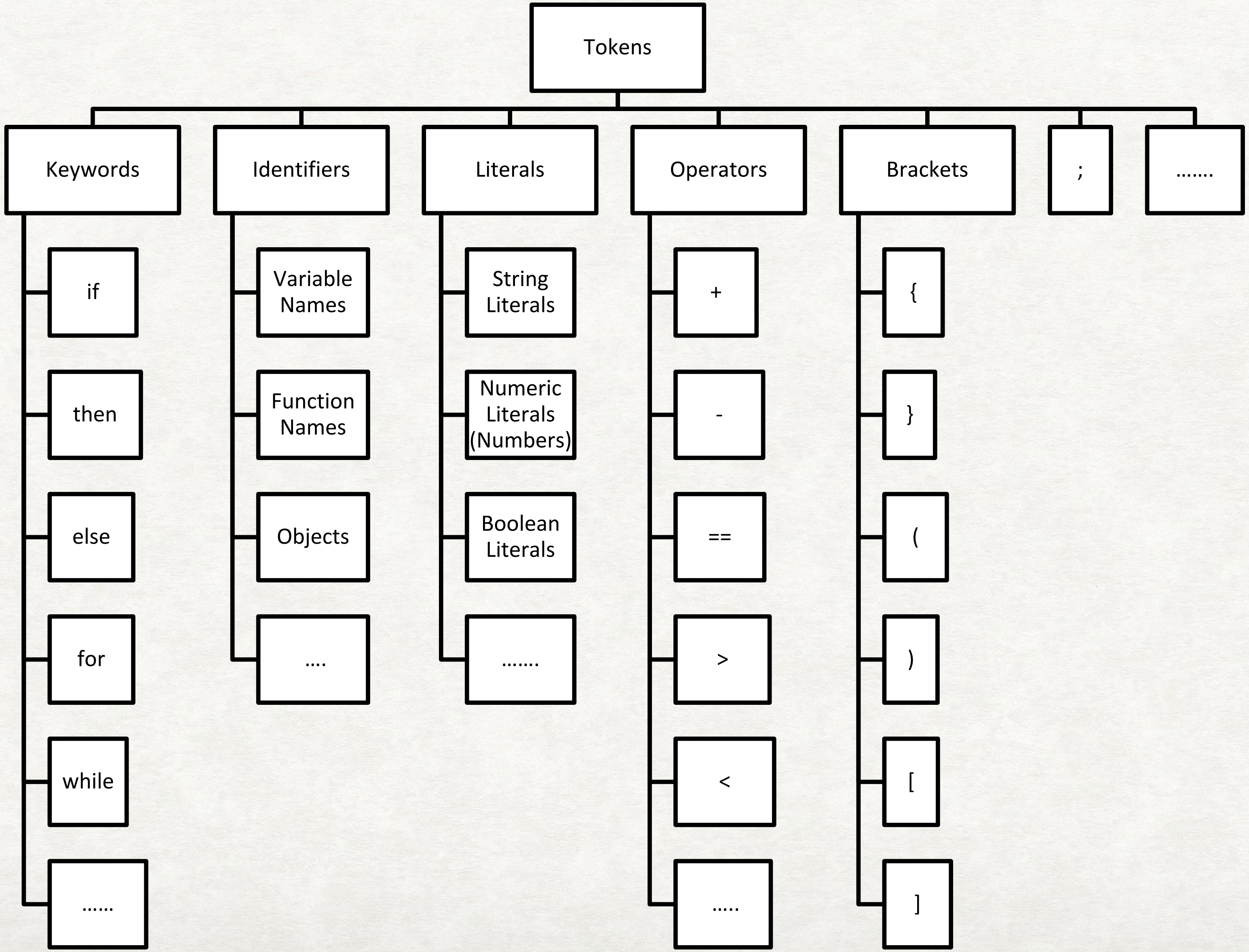


COMMON TOKENS

Numbers	1, -15, 0.1765
Identifiers	Variable names: x, y, z Function names: sum, increment
Operators	+, -, ==
Brackets	(,), {, }
Keywords	if, then, else, for
String literals	"Hello world!"
Whitespace	spaces, newlines, tabs etc (normally not tokens but they are in some languages like Python)

Comments are never tokens

Each token belongs to a Token Class.



PHASE I : LEXICOGRAPHICAL ANALYSIS

The **input** to a scanner is a high-level language statement from the source program.

Its **output** is a list of all the tokens contained in that statement, as well as the classification of each token found.

AN EXAMPLE : INPUT TO THE LEXER

Source Code

```
x = 40;
diff = 2;
if (x + diff == 42) {
/* For those who have read Douglas
Adams */
printf("%d is the meaning of  life", x
+ diff);
}
```

Example Tokens

1. <symbol>
2. <number>
3. <string literal>
4. if
5. printf
6. =
7. ==
8. (
9.)
10. {
11. }
12. ,
13. ;
14. +

Or more …

Output from the Lexer

x	1
=	6
40	2
;	13
diff	1
=	6
2	2
;	13
if	4
(	8
x	1
+	14
diff	1
==	7
42	2
)	9
{	10
printf	5
(	8
"%d is the meaning of life“	3
,	12
x	1
+	14
diff	1
)	9
;	13
}	11

Token Classes

1. <symbol>
2. <number>
3. <string literal>
4. if
5. printf
6. =
7. ==
8. (
9.)
10. {
11. }
12. ,
13. ;
14. +

PRACTICE EXERCISES

Using the token classes as given in the Figure (from the textbook), count how many tokens and token classes do each of the following statements have. What will be the lexer output in each case?

```
limit = begin + end;
```

```
a = b - 1;
```

```
if(c==50) x=1;else y = x+44;
```

FIGURE 11.3

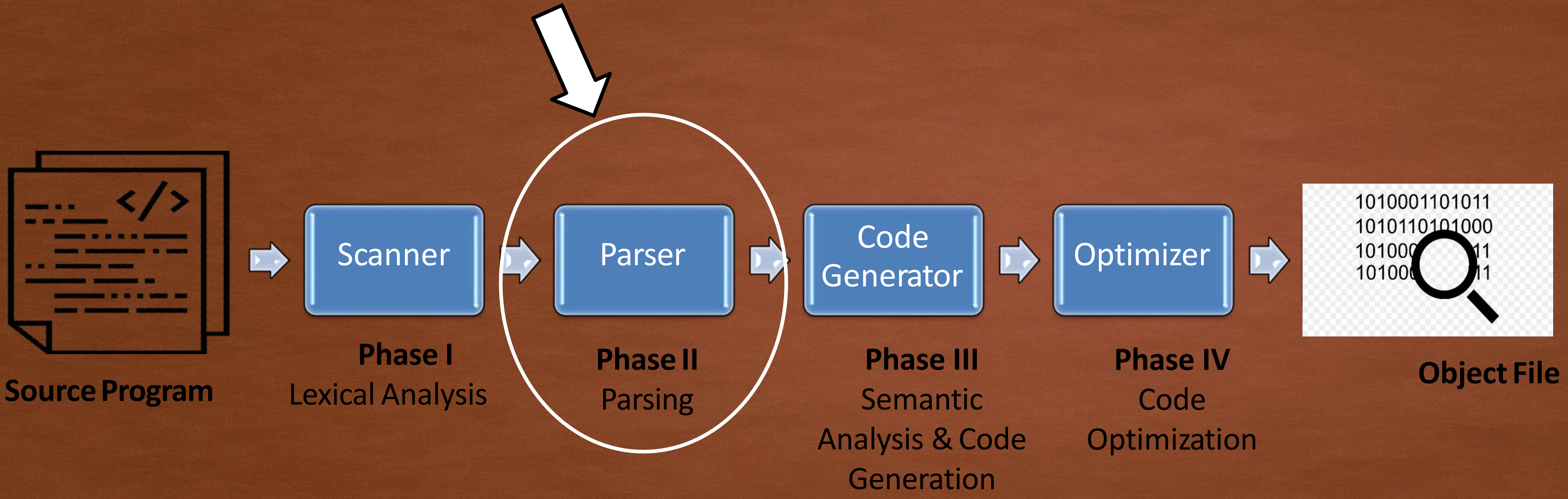
Token Type	Classification Number
symbol	1
number	2
=	3
+	4
–	5
;	6
==	7
if	8
else	9
(	10
)	11

```
limit = begin + end;
```

```
a = b - 1;
```

```
if(c==50) x=1;else y = x+44;
```

Phase II : Parsing: First Principles



Phase II : Parsing

- The sequence of tokens formed by the scanner is checked to see whether it is ***syntactically correct*** according to the ***rules*** of the programming language.

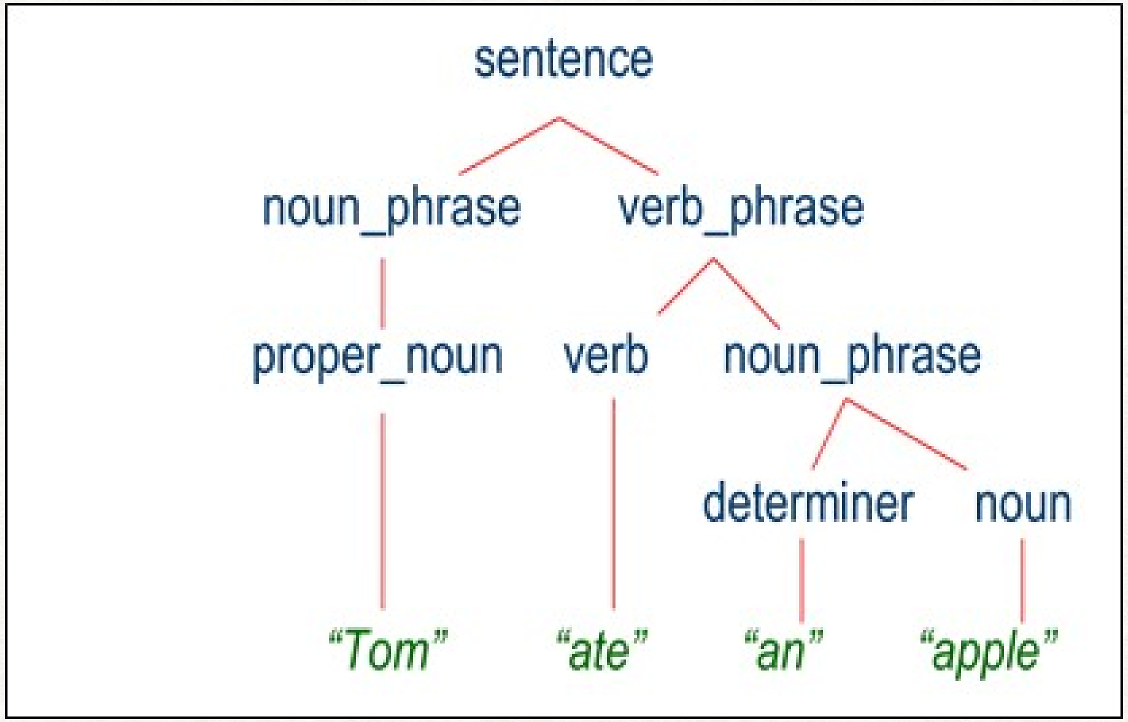
- For this, the parser must be given the rules of the language:
  - a **formal description of the syntax** of the language (or, the **grammar** of the language).

- Input to a Parser
  - A list of classified tokens from the lexical analyser/scanner
  - A grammar specifying the syntax of the language.

- Output of a Parser
  - A parse tree that represents the syntactic structure of a program according to the rules of the given grammar.

Phase II : Parsing

Example – Natural language parsing



sentence -> noun_phrase, verb_phrase
noun_phrase -> proper_noun
noun_phrase -> determiner, noun
verb_phrase -> verb, noun_phrase
proper_noun -> [Tom]
noun -> [apple]
verb -> [ate]
determiner -> [an]

BNF (Backus-Naur Form)

- The most widely used notation for representing the syntax of a programming language is **BNF (Backus- Naur Form).**
  - It is a formal, mathematical way to specify the grammar of a programming language.

The syntax of a language in BNF is specified as a set of ***rules,*** also called (**productions**)

$$\underbrace{\text{left-hand side}}_{\text{Category}} \quad \underbrace{::==}_{\text{“is defined as”}} \quad \underbrace{\text{“definition”}}_{\text{Grammatical structure of the category}}$$

```
1.<symbol>          ::==      x | y | z
2.<expression>      ::==      <symbol> + <symbol>
3.<assignment-stmt> ::==      <symbol> = <expression>
```

This set of ***rules*** is called the ***grammar*** of the language.

`<, >, ::=, |, Λ` : are the metasymbols in the BNF grammar

Terminology: Terminals vs. Non-terminals

- **Terminals**
    - are the actual tokens of the language returned by a lexer.
    - e.g., “+” or “(“ or <symbol> or <number>
    - There is no rule in the grammar that defines a terminal.
- **Non-terminals**
    - A non-terminal is not an actual element of the language but an intermediate grammatical category in the rules of the language.
        - e.g., <expression> or <assignment-statement>
    - Non-terminals are always written in angular brackets < >
- **Goal Symbol**
    - Special nonterminal object that the parser is trying to produce

PARSING

The parse tree has as its leaves the terminals (tokens returned by the lexer).

The parser uses the rules of the grammar to combine terminals into non-terminals, or transform non-terminals into other non-terminals.

Each such step of combining/transforming terminals/non-terminals into other non-terminals is called a *production.*

Eventually, there should only one branch left: the *root* representing the *goal symbol*

If the parser cannot build such a tree, it concludes that the code has a *syntax error*.